18 | 购物车信息确定订单：为什么动态参数化逻辑非常重要?

2021-04-30 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 26:08 大小 23.94M

你好，我是高楼。

我们今天来看一下购物车信息确定订单这个接口的性能怎么样，有哪些需要优化的地方。

在这节课中，我将给你展示如何进行方法级的跟踪，来判断参数的问题。而这个参数，并不是我们这个接口直接用到的，它有不同的使用层次。

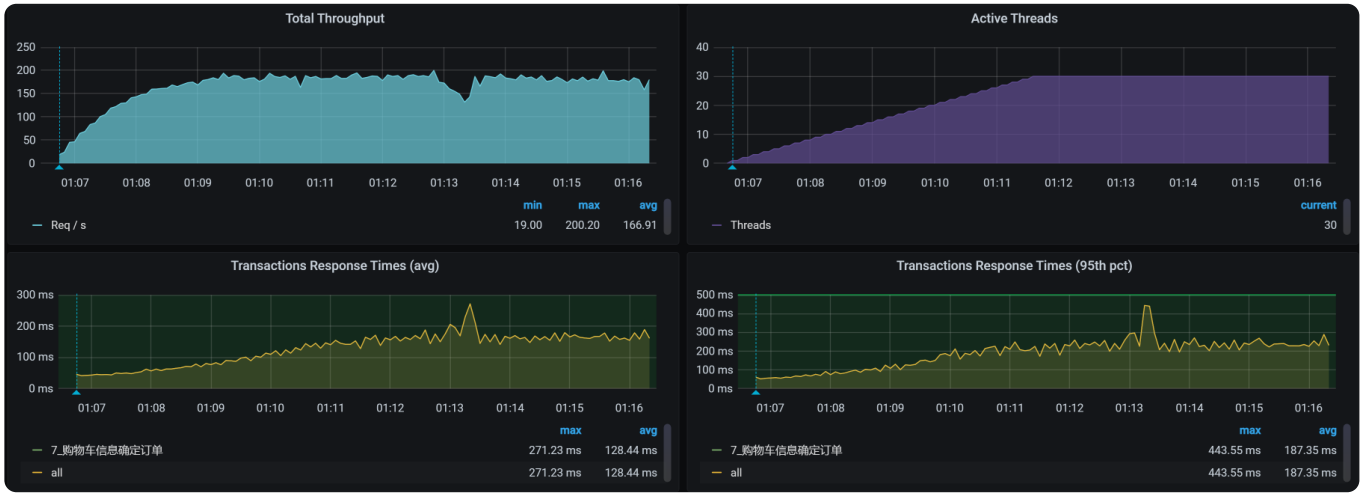
直接的参数化我们都能理解，对吧。但是当一个参数产生新的数据，而新的数据又会在后续的动作中用到时，你就得注意了，因为我们有可能在第一层数据中没有发现问题，但在后续的动作中会遇到问题。所以，我们一定要关注参数化的变化，也就是动态的参数化的数据。

此外，在这节课中，我还将带你一起来看看在应用有多个节点的情况下，某个节点消耗资源过多导致的复杂问题该怎么处理。

话不多说，我们开始今天的分析吧！

场景运行数据

对于购物车信息确定订单这个接口，我们第一次运行的性能场景结果如下：

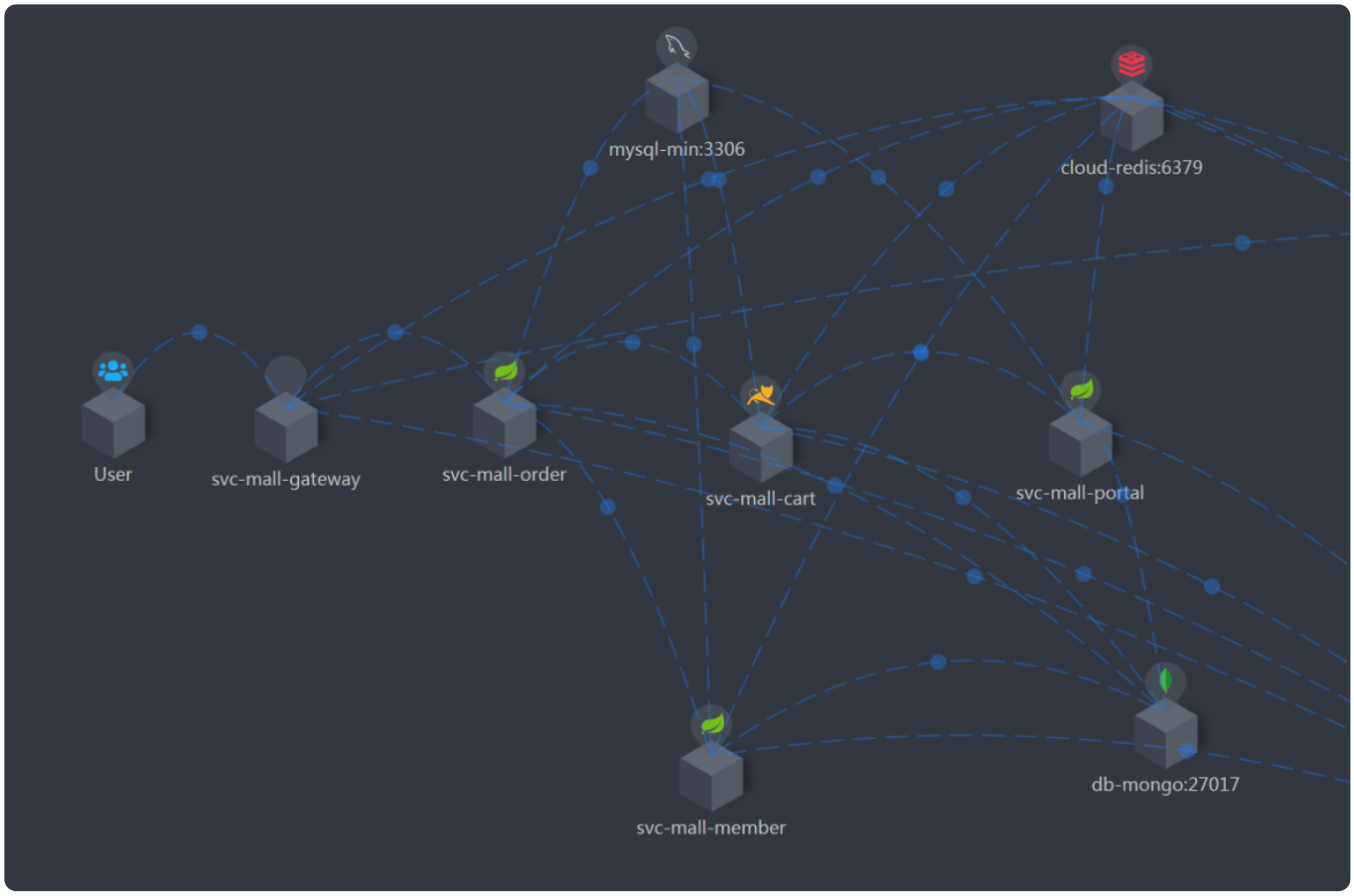


在图中，响应时间随着压力的增加而增加，而 TPS 只到了 160 多，还是有点低了，我们现在就得考虑把 TPS 提升。

注意，**这是一个典型的 TPS 不高，响应时间不断增加的性能问题。**

按照 RESAR 性能分析逻辑，我们看一下这个接口的架构图。

看架构图

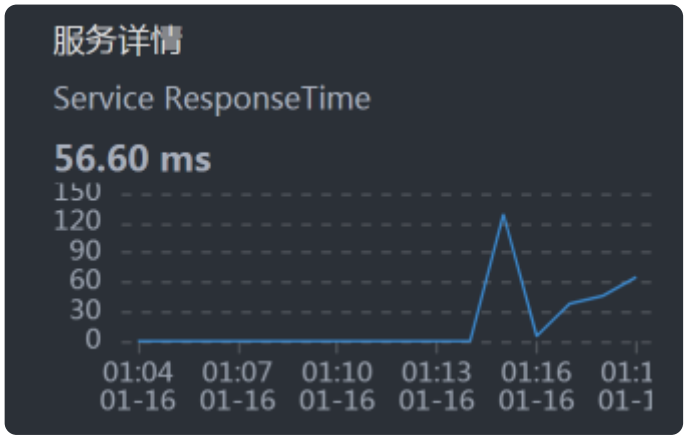


可以看到，这个接口涉及到的服务比较多，架构图也比之前其他接口的要复杂一些。

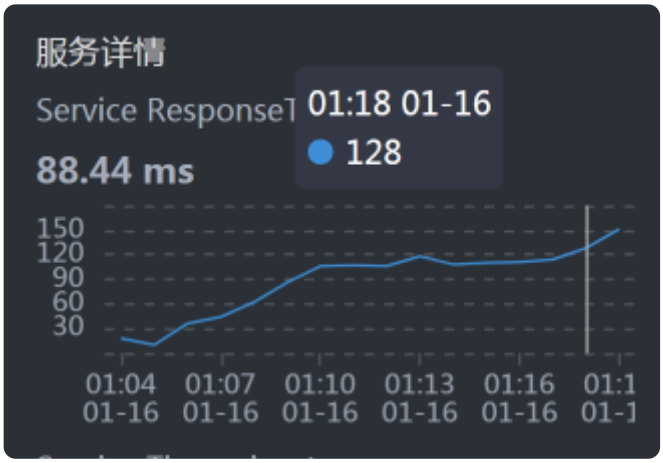
紧接着，我们就来拆分响应时间。

拆分响应时间

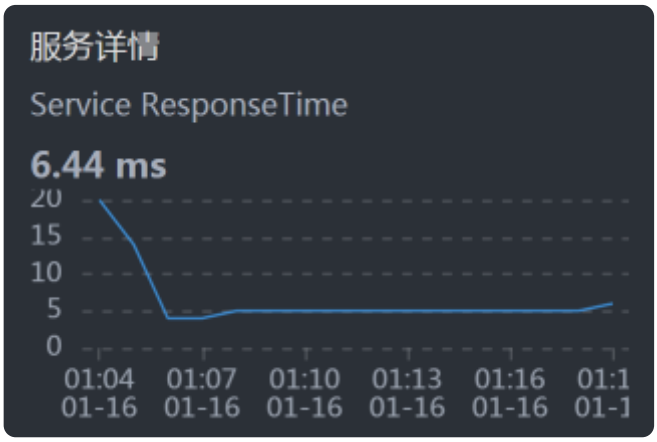
Gateway：



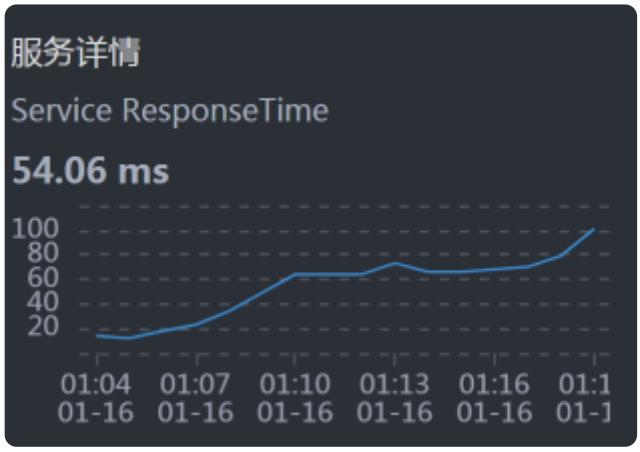
Order：



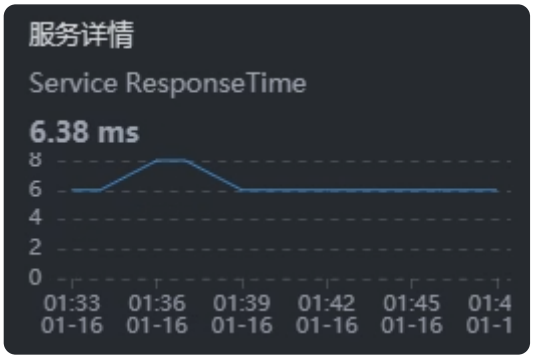
Member：



Cart：

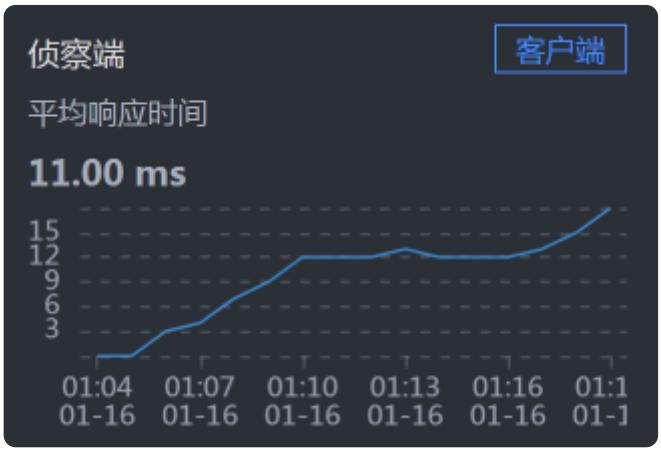


Portal：



从上面的时间拆分来看，Cart 消耗了最长的时间。所以，我们先分析 Cart。

我们再顺手点一下 Cart 和 MySQL 之间的时间消耗，看看是什么情况：



这个 Cart 和 MySQL 之间的时间看起来不长，那我们就不用考虑数据库的 SQL 时间消耗了。

接下来，我们就来分析响应时间长的 Cart 服务。

第一阶段

全局分析

按照惯例，我们来看一下 worker 层面的资源消耗情况：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.30 week	7.64 GiB	4	0.36	11.16%	24.03%	29.95%	0 B/s	86.85 kB/s	290	35	416.98 kbps	450.20 kbps
k8s-master-2	k8s-master-2	3.88 week	7.64 GiB	4	0.80	17.50%	23.17%	29.95%	0 B/s	75.53 kB/s	284	30	249.88 kbps	263.07 kbps
k8s-master-3	k8s-master-3	3.89 week	7.64 GiB	4	0.23	8.30%	24.66%	29.95%	0 B/s	79.42 kB/s	318	42	996.86 kbps	996.86 kbps
k8s-worker-1	k8s-worker-1	2.49 day	15.51 GiB	6	0.63	11.49%	53.40%	44.75%	27.62 kB/s	7.46 kB/s	29	93	323.63 Mbps	342.00 Mbps
k8s-worker-2	k8s-worker-2	2.49 day	15.51 GiB	6	2.15	23.30%	45.99%	29.95%	13.81 kB/s	48.88 kB/s	30	13 K	38.99 Mbps	55.89 Mbps
k8s-worker-3	k8s-worker-3	2.49 day	15.51 GiB	6	17.53	81.05%	44.67%	33.93%	36.82 kB/s	20.38 kB/s	30	645	324.76 Mbps	336.32 Mbps
k8s-worker-4	k8s-worker-4	2.80 week	15.51 GiB	6	0.20	5.49%	22.95%	31.58%	13.81 kB/s	10.14 kB/s	47	101	811.39 kbps	922.36 kbps
k8s-worker-5	k8s-worker-5	2.79 week	15.51 GiB	6	1.59	27.75%	46.75%	36.60%	23.02 kB/s	6.68 kB/s	28	112	9.27 Mbps	15.56 Mbps
k8s-worker-6	k8s-worker-6	2.79 week	15.51 GiB	6	3.34	38.66%	34.74%	39.07%	10.74 kB/s	29.48 kB/s	31	122	27.16 Mbps	27.47 Mbps
k8s-worker-7	k8s-worker-7	2.49 week	15.51 GiB	4	0.33	9.47%	40.62%	40.89%	13.81 kB/s	4.98 kB/s	28	144	19.95 Mbps	20.01 Mbps
k8s-worker-8	k8s-worker-8	1.32 week	15.51 GiB	4	4.60	55.21%	30.14%	44.32%	0 B/s	4.51 kB/s	29	478	43.32 Mbps	32.31 Mbps
k8s-worker-9	k8s-worker-9	2.49 week	15.51 GiB	6	0.88	12.21%	41.41%	29.95%	4.60 kB/s	2.96 kB/s	28	79	15.27 Mbps	29.22 Mbps

从上图来看，worker-3 上消耗的资源较多。那我们就来查看一下 worker-3 上有什么服务。

复制代码

```
[root@k8s-master-2 ~]# kubectl get pods -o wide | grep k8s-worker-3
cloud-nacos-registry-685b8957d7-vskb6       1/1     Running     0          2d1
cloud-redis-7f7db7f45c-t5g46                2/2     Running     0          2d8
elasticsearch-master-2                      1/1     Running     0          3h2
svc-mall-cart-558d787dc7-g6qgh              1/1     Running     0          2d1
svc-mall-order-fbfd8b57c-kbczh              1/1     Running     0          2d1
svc-mall-portal-846d9994f8-m7jbq            1/1     Running     0          38h
svc-mall-search-c9c8bc847-h7sgv             1/1     Running     0          161
[root@k8s-master-2 ~]#
```

可以看到，worker-3 上有 8 个服务，哪个服务消耗的资源最多呢？现在我们进入 worker-3，查看下 top：

复制代码

```
[root@k8s-worker-3 ~]# top
top - 01:51:35 up 2 days, 12:18,  2 users,  load average: 19.48, 18.40, 17.07
Tasks: 319 total,   1 running, 318 sleeping,   0 stopped,   0 zombie
%Cpu0  : 68.6 us,  6.4 sy,  0.0 ni, 19.9 id,  0.0 wa,  0.0 hi,  5.1 si,  0.0 s
%Cpu1  : 66.7 us,  5.8 sy,  0.0 ni, 22.8 id,  0.0 wa,  0.0 hi,  4.8 si,  0.0 s
%Cpu2  : 66.4 us,  6.1 sy,  0.0 ni, 22.7 id,  0.0 wa,  0.0 hi,  4.7 si,  0.0 s
%Cpu3  : 65.7 us,  5.4 sy,  0.0 ni, 23.6 id,  0.0 wa,  0.0 hi,  5.4 si,  0.0 s
%Cpu4  : 66.6 us,  5.7 sy,  0.0 ni, 22.0 id,  0.0 wa,  0.0 hi,  5.7 si,  0.0 s
%Cpu5  : 67.6 us,  5.8 sy,  0.0 ni, 22.5 id,  0.0 wa,  0.0 hi,  4.1 si,  0.0 s
KiB Mem : 16265992 total,  2525940 free,  7015104 used,  6724948 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  8848464 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32216 root      20   0 8878548 658820  16980 S 280.5  4.1 375:31.82 java -Dapp
32589 root      20   0 8839408 589196  15892 S  84.1  3.6 171:16.88 java -Dapp
24119 root      20   0 8798548 549804  15892 S  65.9  3.4 115:52.74 java -Dapp
 1089 root      20   0 2438956 105708  37776 S   6.3  0.6 248:21.71 /usr/bin/k
```

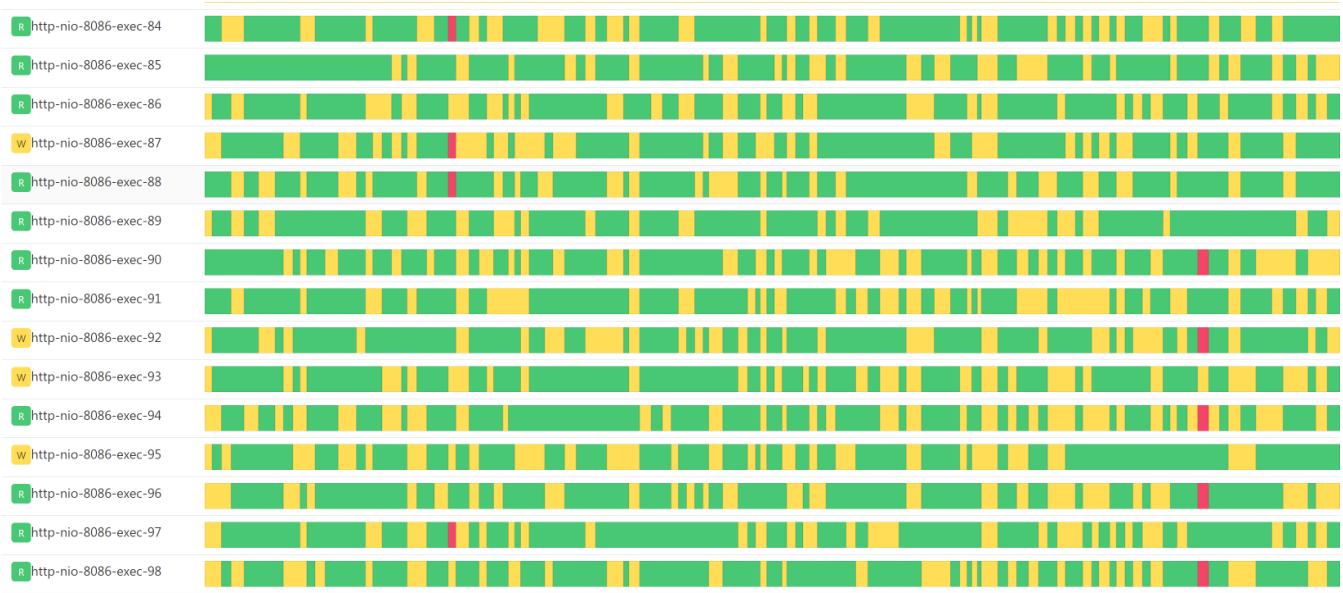
```
19  5470 root      20   0 1154816  14992   1816 S   3.6  0.1  20:15.93 redis-serv
```

从以上数据来看，的确是 Cart 服务消耗的 CPU 比较高。不过，它**还没有把 6 个 CPU 都用完**，这一点我们要记一下。

下面开始定向分析。

定向分析

既然 Cart 服务消耗的 CPU 多，那我们当然要看一下 Cart 中的线程都在干啥。



这些线程状态基本都在绿色的 Runnable 状态，看起来都比较繁忙，有可能是因为线程数配置的太低了，我们查看下配置：

复制代码

```
server:
  port: 8086
  tomcat:
    accept-count: 1000
    threads:
      max: 20
      min-spare: 5
    max-connections: 500
```

知道了 Spring Boot 内置的 Tomcat 线程数配置，我们拆分一下在 Cart 上正在执行的方法，看看我们的定位方法是不是合理：

看这张图的时候，你要注意消耗时间长的位置，也就是图中右侧线段比较长的地方。这里面有两个环节的问题：

1. MySQL 的执行时间长。你要注意哦，虽然这里的 MySQL/JDBI/PreparedStatement/execute 并没有消耗很长的时间，但是它的下一步 Balance/Promotion/Cart/CartPromotion 消耗的时间是长的；

2. Promotionnew 方法本身的时间长。

由于慢的节点和 MySQL 有关，我们创建一个，mysqlreport 来看 MySQL 整体的监控数据：

```
__ Connections _____________________________________________________________
Max used          152 of  151      %Max: 100.66
Total             540     0.0/s
```

原来是连接用完了！我们赶紧改一下，从 151 改到 500。

不过，重测之后响应时间还是没有变化，那我们就只能接着跟踪 Cart 上的方法了。

方法级跟踪

于是，我们不得不来到方法级跟踪， 看一下我们关注的方法 Promotionnew 慢在哪里。

由上面那张调用视图，我们可以编写下面这样的跟踪语句：

trace -E com.dunshan.mall.cart.controller.CartItemController listPromotionnew -n 5 -v --skipJDKMethod false '1==1'

然后得到了下面这个结果：

```
`---ts=2021-01-16 15:08:58;thread_name=http-nio-8086-exec-34;id=f8;is_daemon=t
    `---[97.827186ms] com.dunshan.mall.cart.service.imp.CartItemServiceImpl$$E
        `---[97.750962ms] org.springframework.cglib.proxy.MethodInterceptor:in
            `---[97.557484ms] com.dunshan.mall.cart.service.imp.CartItemServic
                +---[72.273747ms] com.dunshan.mall.cart.service.imp.CartItemSe
                +---[0.003516ms] cn.hutool.core.collection.CollUtil:isNotEmpty
                +---[0.004207ms] java.util.List:stream() #173
                +---[0.003893ms] java.util.stream.Stream:filter() #57
                +---[0.003018ms] java.util.stream.Collectors:toList() #57
                +---[0.060052ms] java.util.stream.Stream:collect() #57
                +---[0.002017ms] java.util.ArrayList:<init>() #177
                +---[0.003013ms] org.springframework.util.CollectionUtils:isEm
                `---[25.152532ms] com.dunshan.mall.cart.feign.CartPromotionSer
```

可以看到，在我们跟踪的方法 com.dunshan.mall.cart.service.imp.CartItemServiceImpl:listPromotion() 中，有两处 listPromotion 和 calcCartPromotion 时间消耗较大，分别是：

1. com.dunshan.mall.cart.service.imp.CartItemServiceImpl:list()
2. com.dunshan.mall.cart.feign.CartPromotionService:calcCartPromotion()

跟踪 List 函数

我们在 Arthas 中执行 trace 跟踪语句如下：

```
1 trace com.dunshan.mall.cart.service.imp.CartItemServiceImpl list -v -n 5 --ski
```

复制代码

然后得到这样的结果：

```
1 `---ts=2021-01-16 15:19:45;thread_name=http-nio-8086-exec-65;id=23ce;is_daemon
2     `---[70.158793ms] com.dunshan.mall.cart.service.imp.CartItemServiceImpl:li
3         +---[0.003501ms] com.dunshan.mall.model.OmsCartItemExample:<init>() #1
4         +---[0.002642ms] com.dunshan.mall.model.OmsCartItemExample:createCrite
5         +---[0.002932ms] com.dunshan.mall.model.OmsCartItemExample$Criteria:an
6         +---[0.00304ms] com.dunshan.mall.model.OmsCartItemExample$Criteria:and
7         `---[70.078976ms] com.dunshan.mall.mapper.OmsCartItemMapper:selectByEx
```

复制代码

在一阵无聊的 trace 之后，看到一个 select 语句消耗时间较长，这个 select 语句是：

```
1 select id, product_id, product_sku_id, member_id, quantity, price, product_pic
```

复制代码

一个简单的 select 语句，怎么会耗时这么久呢？我们先不管为什么会这样，先来看看这个 oms_cart_item 的数据有多少。我连上数据库后一查，发现在 oms_cart_item 里面有 10612 条数据，这个数据量并不大。

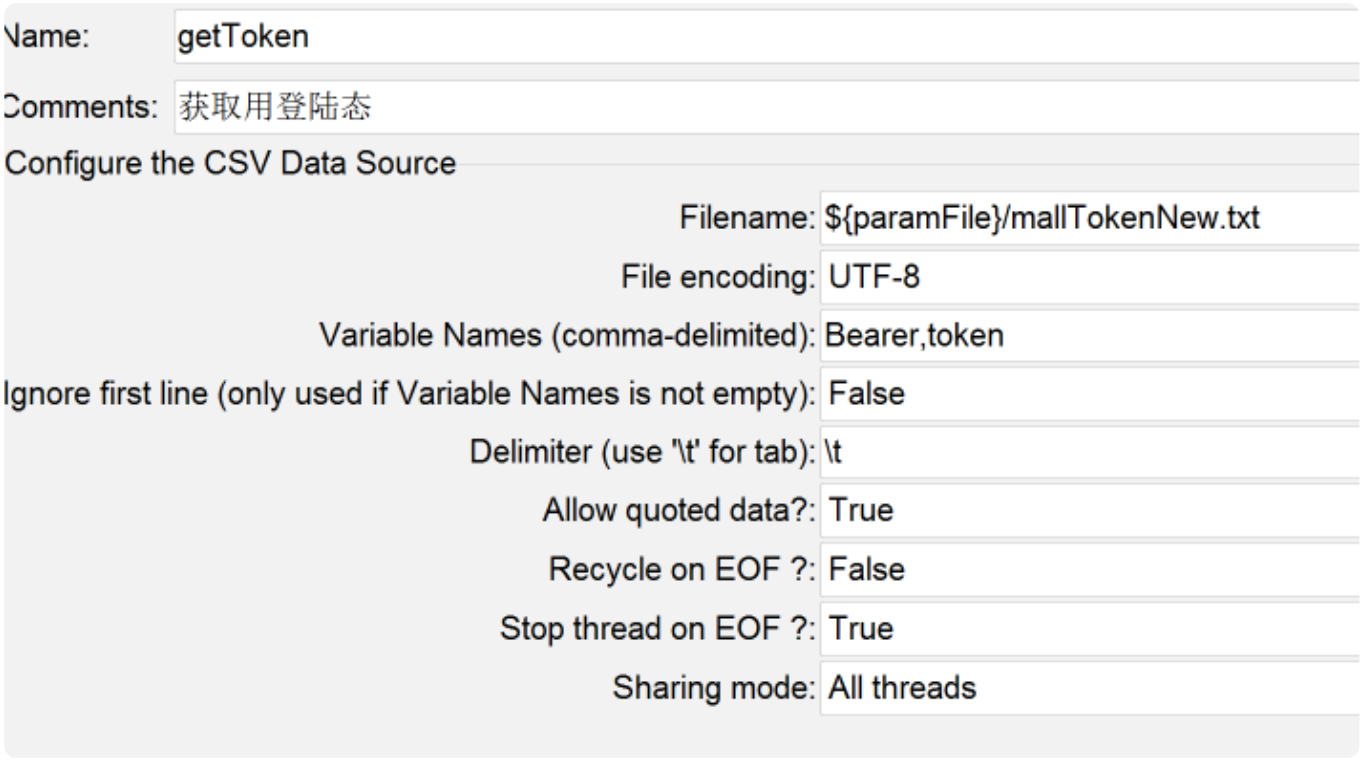
此外，我还查看了一下索引，也是有的，并且执行计划也走到了索引这里。那为什么会慢呢？到这里，我们得考虑一下是不是和数据量有关了。所以，我们来看看这个 select 语句究竟查出了多少条数据。

在我补全 SQL 后一查，发现一个 member_id 对应 500 多条记录，这是一个人一次买了 500 个东西?

既然在购物车信息里，同一个人有这么多记录，那一定是在商品加入购物车时加进去的。而要往一个人的购物车里加东西，显然是在性能脚本执行的过程中添加，因为每个用户的购物车一开始都是空的。所以，我们要去查一下商品加入购物车的脚本是怎么回事。

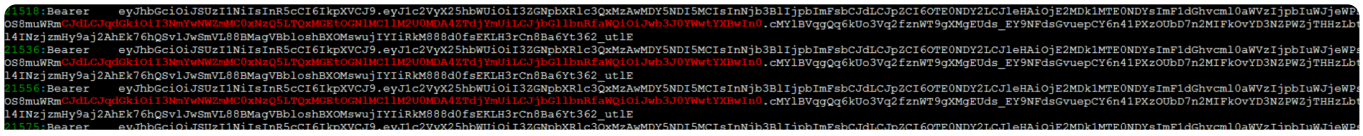
商品加入购物车的脚本很简单，就是一个 post，加上商品 ID，在 HTTP 协议的请求头里面有一个 Token 来标识是哪个用户。

在这里，我们要查的就是 token 有没有用重，JMeter 中的 Token 参数化配置如下：


Name: getToken
Comments: 获取用登陆态
Configure the CSV Data Source
Filename: ${paramFile}/mallTokenNew.txt
File encoding: UTF-8
Variable Names (comma-delimited): Bearer,token
Ignore first line (only used if Variable Names is not empty): False
Delimiter (use '\t' for tab): \t
Allow quoted data?: True
Recycle on EOF ?: False
Stop thread on EOF ?: True
Sharing mode: All threads

看起来挺好，我们在这里设计了不重用数据，所以在设置上 Token 不会被重用。那么，只有一种可能，就是 Token 重了。

```
[root@vm-jmeter param]# grep -rin eyJ1c2VyX25hbWUiOiI32GNpbXRlc3QxMzAwMTA2Mjc1OCIsInNjb3BlIjpbImFsbCJdLCJpZCI6NTQxNjc1LCJleHAiOjE2MDk1MTE0NDcsImF1dGhvcml0aWVzIjpbIuWJjeWPsOS8muWRm mallTokenNew.txt | wc -l
34613
[root@vm-jmeter param]# grep CJdLCJqdGkiOiI3NmYwNW2mMC0xNzQ5LTQxMGEtOGNlMC1lM2U0MDA4ZTdjYmUiLCJjbGllbnRfaWQiOiJwb3J0YWwtYXBwIn0 mallTokenNew.txt  | wc -l
34756
[root@vm-jmeter param]#
```

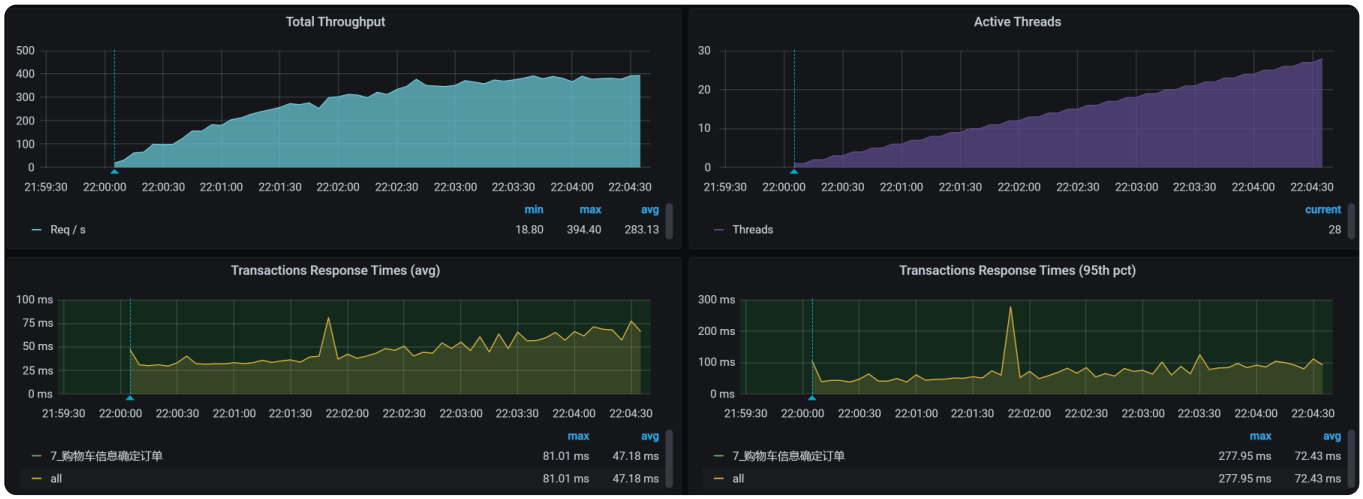


在随机检查了几条 Token 之后，我发现有大量的 Token 重复。这也就解释了，为什么我们会在一个人的购物车里看到那么多商品数据。

可是，这个逻辑就有问题了。你想想，我们设置了参数化中数据不重复使用，但实际上确实有大量的 Token 被重用，这就说明 Token 的参数化文件本身就重复了。

那怎么办呢？我们只有把所有的 Token 全部清掉，让 Token 数据在商品加入购物车的时候不会被重用，以此来避免在一个人的购物车中加入太多商品的情况。

接着怎么办？只有一招了，就是把所有的数据都清掉，然后用所有的用户创建合理的购物车数据。于是，我在这里又花了几个小时，造出了 130 多万数据。现在，我们再回归一下场景：



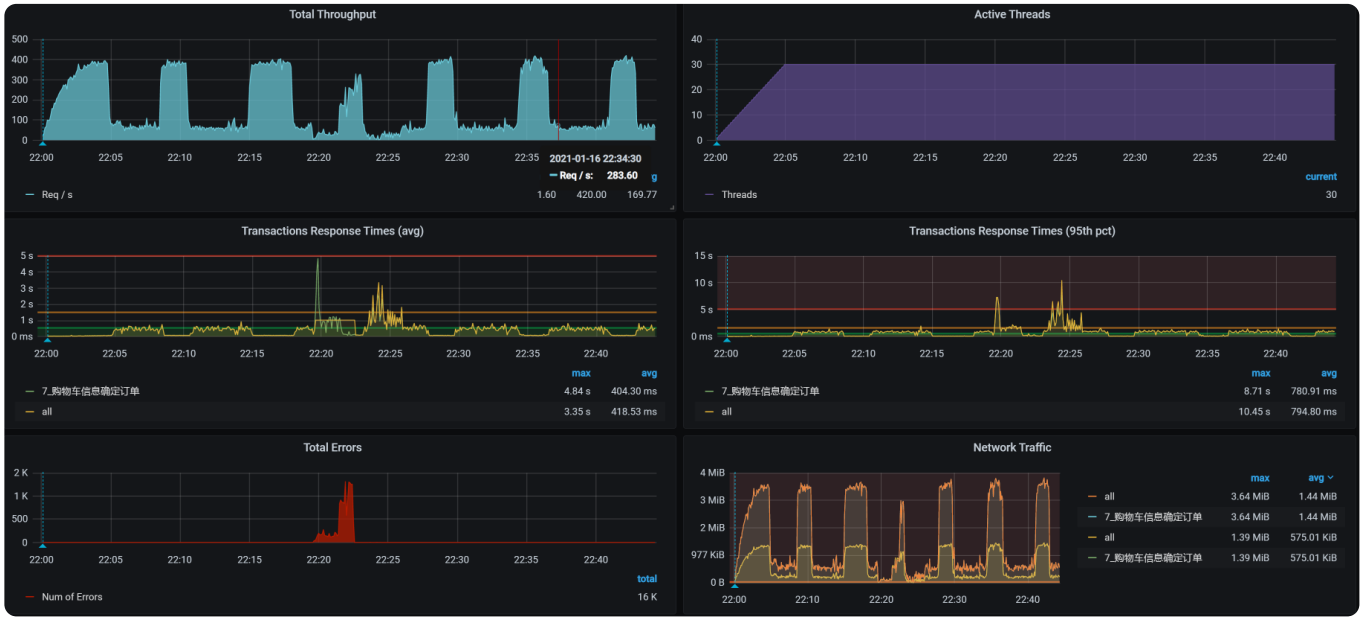
你看，TPS 增加到了 300！

本来这是一个美好到可以喝下午茶的结果，然而……天不随人愿，我在场景持续执行的过程中，又发现了问题，这让我们不得不开启第二阶段的分析。

第二阶段

场景运行数据

是什么问题呢？我在压力运行的数据中，竟然看到了这种 TPS 曲线：

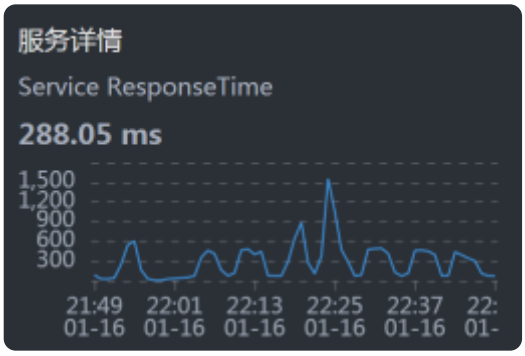


你看，TPS 相当规律地往下掉，不仅会掉下去，而且还会恢复回去，形成一个明显的锯齿状，并且这锯齿还挺大。

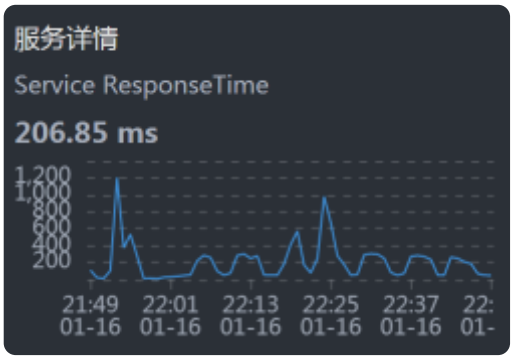
怎么办？根据高老师的思路，现在我们就得按照 RESAR 性能分析逻辑来收拾这个问题了。我们在前面已经看过架构图，所以，现在直接来拆分响应时间。

拆分响应时间

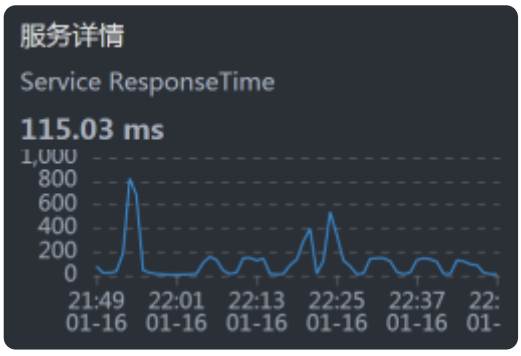
Gateway：



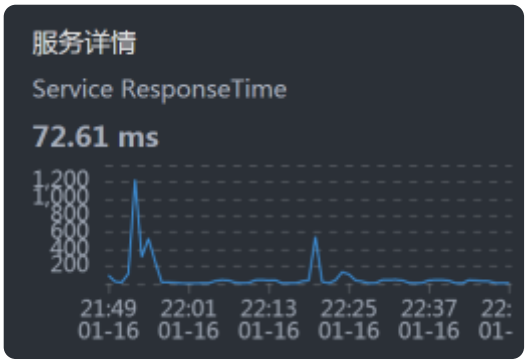
Order：



Cart：



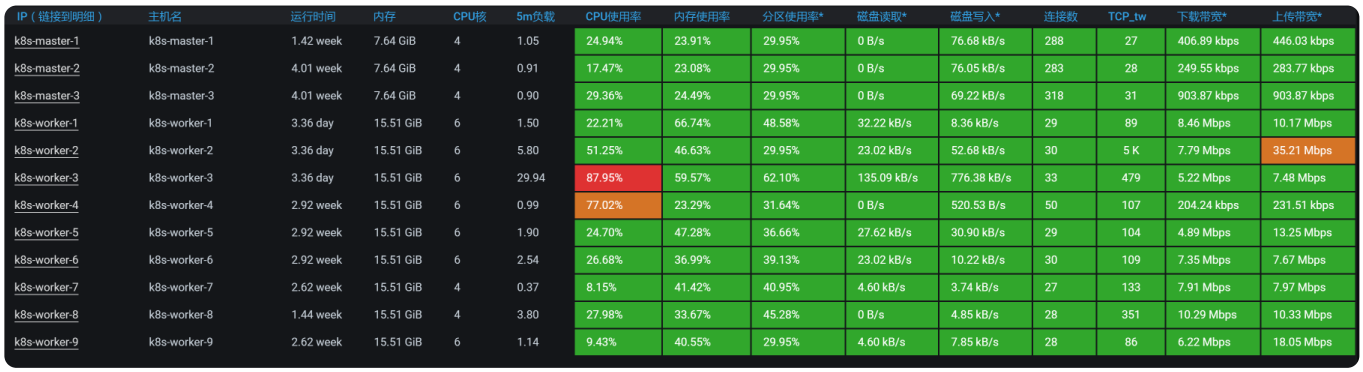
Portal：



从上面的数据来看，似乎每一层都有时间消耗，性能都不怎么样。

全局分析

那我们就查看下当前的全局监控数据，可以看到 worker-3 上的 CPU 消耗最多：



IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.42 week	7.64 GiB	4	1.05	24.94%	23.91%	29.95%	0 B/s	76.68 kB/s	288	27	406.89 kbps	446.03 kbps
k8s-master-2	k8s-master-2	4.01 week	7.64 GiB	4	0.91	17.47%	23.08%	29.95%	0 B/s	76.05 kB/s	283	28	249.55 kbps	283.77 kbps
k8s-master-3	k8s-master-3	4.01 week	7.64 GiB	4	0.90	29.36%	24.49%	29.95%	0 B/s	69.22 kB/s	318	31	903.87 kbps	903.87 kbps
k8s-worker-1	k8s-worker-1	3.36 day	15.51 GiB	6	1.50	22.21%	66.74%	48.58%	32.22 kB/s	8.36 kB/s	29	89	8.46 Mbps	10.17 Mbps
k8s-worker-2	k8s-worker-2	3.36 day	15.51 GiB	6	5.80	51.25%	46.63%	29.95%	23.02 kB/s	52.68 kB/s	30	5 K	7.79 Mbps	35.21 Mbps
k8s-worker-3	k8s-worker-3	3.36 day	15.51 GiB	6	29.94	87.95%	59.57%	62.10%	135.09 kB/s	776.38 kB/s	33	479	5.22 Mbps	7.48 Mbps
k8s-worker-4	k8s-worker-4	2.92 week	15.51 GiB	6	0.99	77.02%	23.29%	31.64%	0 B/s	520.53 B/s	50	107	204.24 kbps	231.51 kbps
k8s-worker-5	k8s-worker-5	2.92 week	15.51 GiB	6	1.90	24.70%	47.28%	36.66%	27.62 kB/s	30.90 kB/s	29	104	4.89 Mbps	13.25 Mbps
k8s-worker-6	k8s-worker-6	2.92 week	15.51 GiB	6	2.54	26.68%	36.99%	39.13%	23.02 kB/s	10.22 kB/s	30	109	7.35 Mbps	7.67 Mbps
k8s-worker-7	k8s-worker-7	2.62 week	15.51 GiB	4	0.37	8.15%	41.42%	40.95%	4.60 kB/s	3.74 kB/s	27	133	7.91 Mbps	7.97 Mbps
k8s-worker-8	k8s-worker-8	1.44 week	15.51 GiB	4	3.80	27.98%	33.67%	45.28%	0 B/s	4.85 kB/s	28	351	10.29 Mbps	10.33 Mbps
k8s-worker-9	k8s-worker-9	2.62 week	15.51 GiB	6	1.14	9.43%	40.55%	29.95%	4.60 kB/s	7.85 kB/s	28	86	6.22 Mbps	18.05 Mbps

因此，我们来查一下 worker-3 上有哪些 Pod：

复制代码

```
[root@k8s-master-3 ~]# kubectl get pods -o wide | grep k8s-worker-3
cloud-nacos-registry-685b8957d7-vskb6       1/1     Running     0          3d7
cloud-redis-7f7db7f45c-t5g46                2/2     Running     1          3d4
elasticsearch-master-2                      1/1     Running     0          23h
svc-mall-cart-79c667bf56-j76h6              1/1     Running     0          20h
svc-mall-order-fbfd8b57c-kbczh              1/1     Running     0          3d7
svc-mall-portal-846d9994f8-m7jbq            1/1     Running     0          2d1
svc-mall-search-c9c8bc847-h7sgv             1/1     Running     0          23h
[root@k8s-master-3 ~]#
```

居然有这么多服务都在 worker-3 上。

我们现在登录到 worker-3 上，看一下 top 资源。其实，我在这里主要想看的是 process table，因为我想先确定一下哪个服务消耗的资源最高，然后再决定收拾哪个服务。

复制代码

```
[root@k8s-worker-3 ~]# top
top - 22:13:01 up 3 days,  8:39,  3 users,  load average: 40.34, 30.03, 18.02
Tasks: 326 total,   6 running, 320 sleeping,   0 stopped,   0 zombie
%Cpu0  : 74.5 us, 13.4 sy,  0.0 ni,  7.7 id,  0.0 wa,  0.0 hi,  4.4 si,  0.0 s
%Cpu1  : 66.3 us, 12.1 sy,  0.0 ni, 16.5 id,  0.0 wa,  0.0 hi,  4.7 si,  0.3 s
%Cpu2  : 49.7 us, 32.4 sy,  0.0 ni, 14.9 id,  0.0 wa,  0.0 hi,  2.7 si,  0.3 s
%Cpu3  : 73.2 us,  9.7 sy,  0.0 ni, 12.4 id,  0.0 wa,  0.0 hi,  4.7 si,  0.0 s
%Cpu4  : 76.4 us, 10.5 sy,  0.0 ni,  8.8 id,  0.0 wa,  0.0 hi,  4.1 si,  0.3 s
%Cpu5  : 62.4 us, 16.4 sy,  0.0 ni, 16.1 id,  0.0 wa,  0.0 hi,  4.7 si,  0.3 s
KiB Mem : 16265992 total,   211212 free,  9204800 used,  6849980 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6650068 avail Mem
```

```
12
13
14   PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
15 32485 root      20   0 8895760 700564  16896 S 101.6  4.3 723:03.52 java -Dapp
16 32589 root      20   0 8845576 778684  15896 S  93.6  4.8 427:04.44 java -Dapp
17 24119 root      20   0 8825208 600956  15888 S  67.9  3.7 262:00.83 java -Dapp
18 ............
```

在上述 top 资源中，我们主要来看几个吃 CPU 的大户。不难发现，Cart/Order/Portal 这三个服务在购物车信息确定订单的业务中都有用到，并且都在同一个 worker 上。同时，我们也可以看到，在这个 6C 的 worker 中，现在的 CPU 队列已经达到 40 了。

定向分析

从系统上来看，CPU 队列长的问题主要是由于资源的争用，因为线程在不断地唤醒，通过 start_thread 就能看出来：

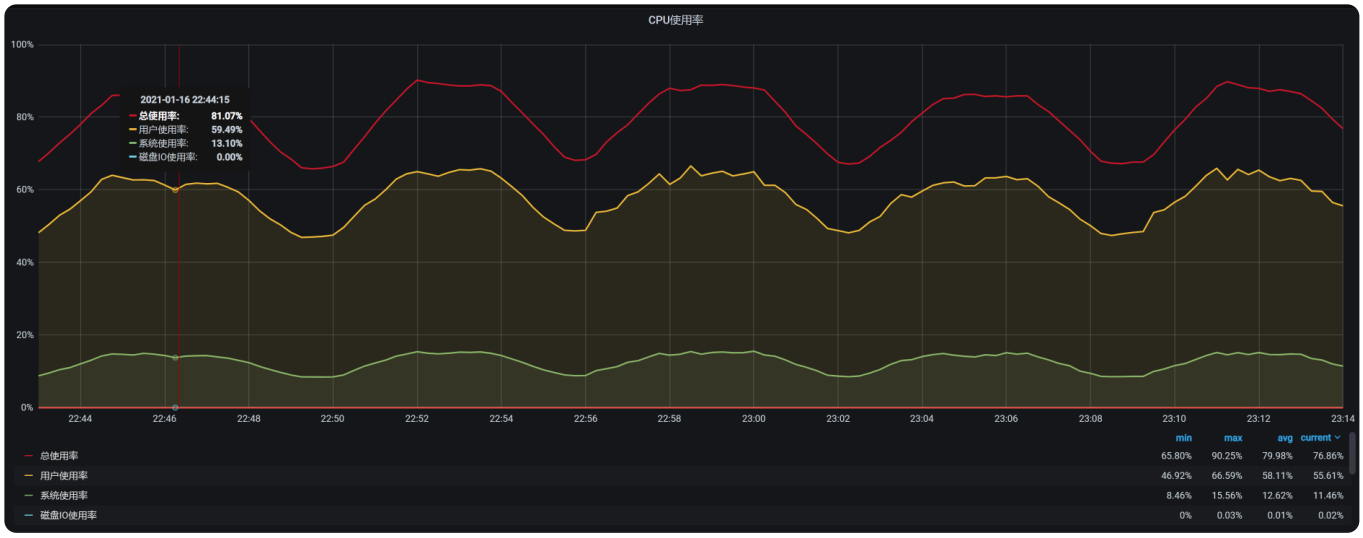
现在我们要做的就是把线程降下去。

怎么降呢？有两种手段：

1. 把服务移走，先一个个收拾，分而治之。
2. 把服务的线程减少。

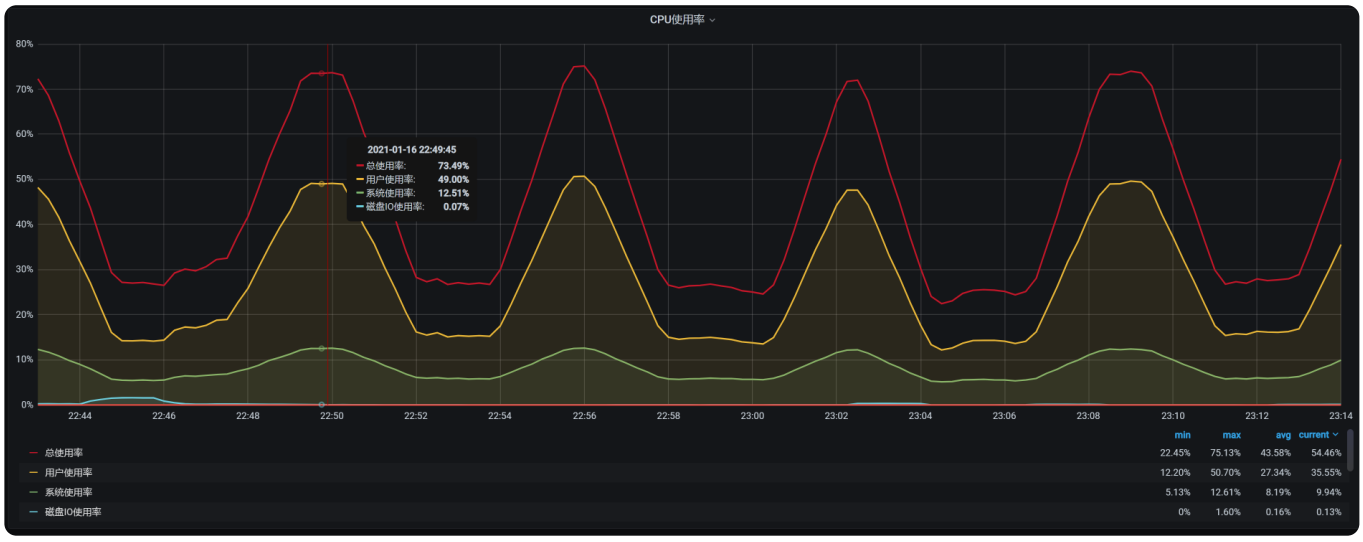
这两种方式都是能减少资源争用，但是会带来不同的影响。其中，第一种手段比较合理，只是会消耗更多的整体资源；第二种手段虽然会减少争用，但是会导致响应时间增加。

我这么跟你一讲，你可能已经发现了，这两种手段都不能解释 TPS 不稳定的问题。那为什么 TPS 一会儿掉下去，一会儿又恢复呢？现在我们还不知道答案，不过基于我们“全局 - 定向”的分析思路，我们先看一下 worker-3 的资源消耗：

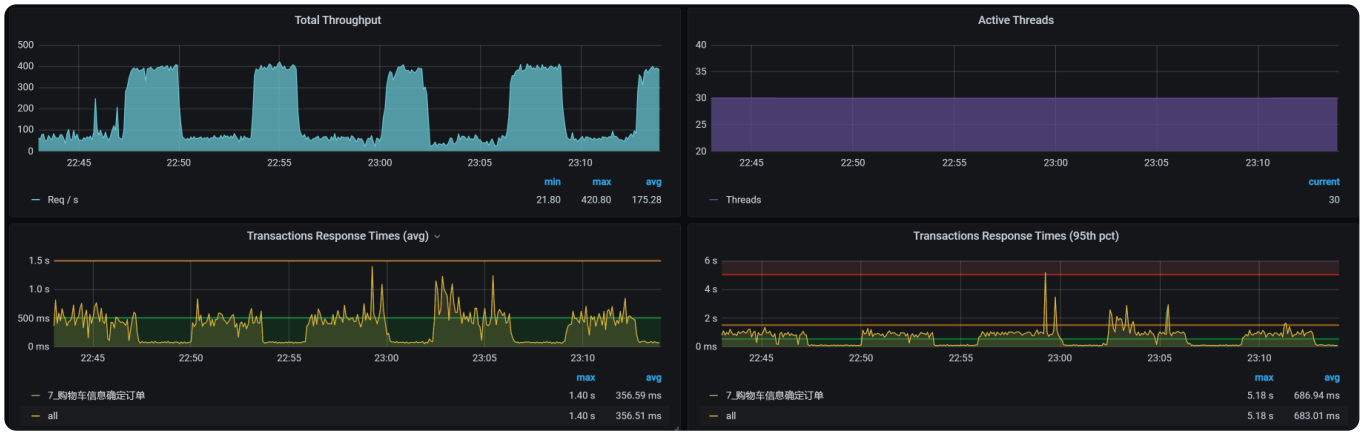


在同一时间段，我也查看了同一台物理机上的其他 worker 的资源消耗情况，发现 worker-8 的资源消耗有点不太正常。请你注意，我此时的查看逻辑，仍然依据的是第 3 讲中描述的逻辑，以及对应第 4 讲中的性能分析决策树。希望你不要觉得这里有跳跃，实际上我们还是在全局监控的第一层计数器上。

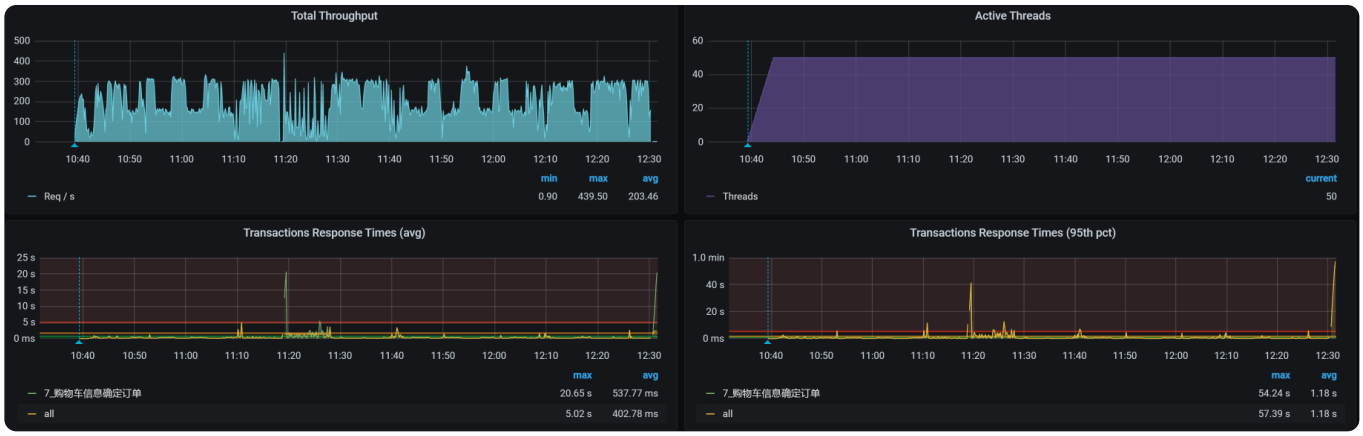
我们具体来看一下 worker-8 的资源消耗：



再来看一下压力场景的执行数据：

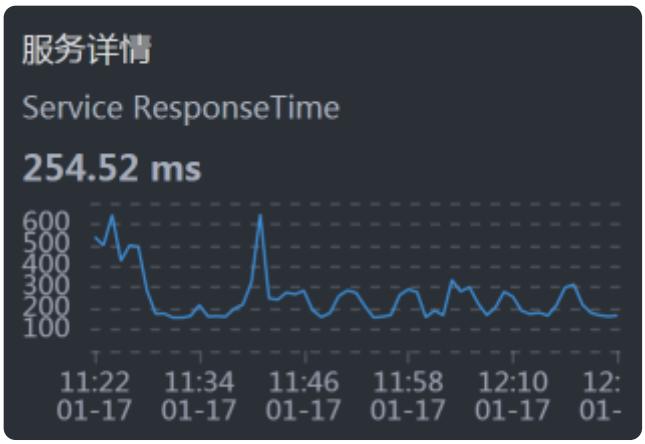


从上面 worker-8 的资源使用率来看，确实有很高的时候。考虑到同一物理机上资源的争用问题，我们现在把 cart 移到 Worker-7 上去，把 order 移到 worker-9 上去，再来看 TPS：

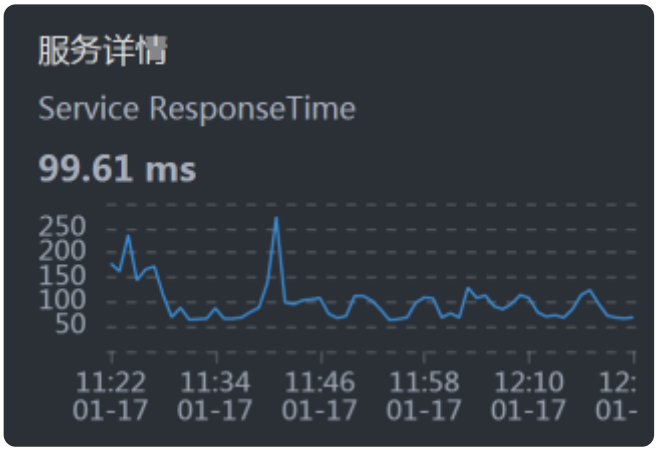


花花绿绿，起起伏伏，真是好看……我们先不管这样做有没有增加 TPS，光看这个趋势，就足以让人心碎了。既然结果还是这样，那我们就用老套路，继续拆分时间来看看。

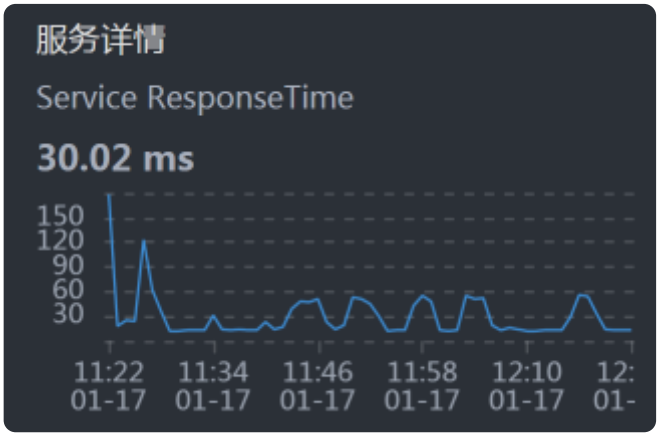
Gateway：



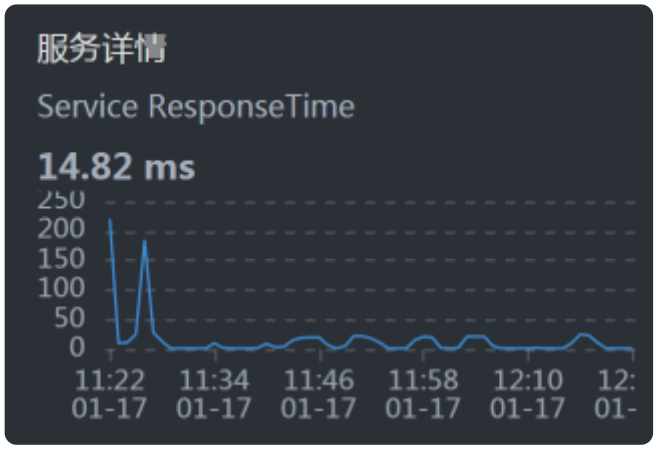
Order：



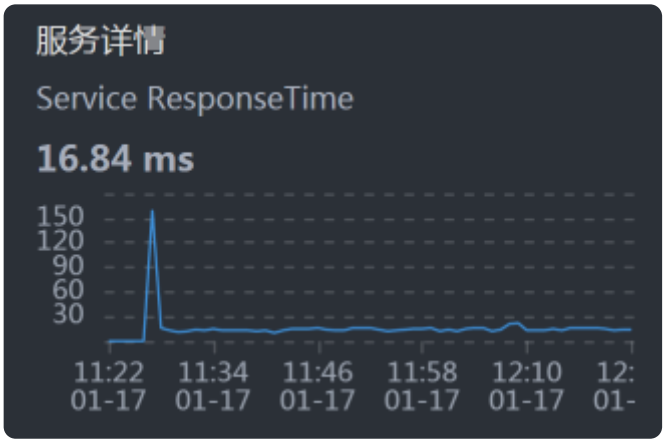
Cart：



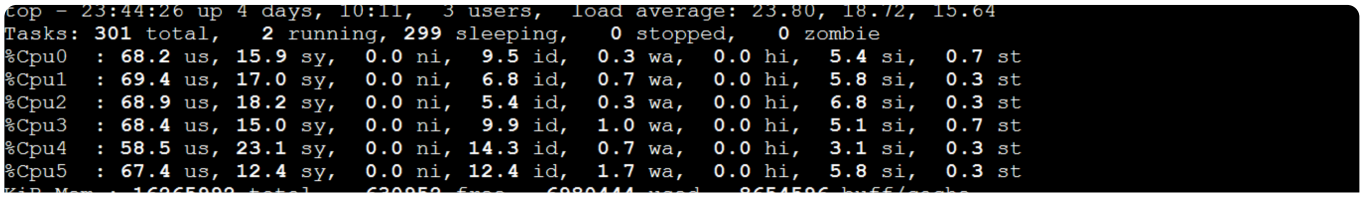
Portal：



Member：



从上面的时间来看，Gateway 消耗的时间比较长，这就奇怪了，这都换了到了 Gateway 服务上有问题了。所以，我们到 Gateway 机器上看一下到底有哪些服务：



呀呀呀，那个占第一位的是个啥？原来是 ES 的一个进程，这个进程消耗了多个 CPU。看到这，我想起来前几天为了增加 ES 的能力，我们给 ES data 和 ES client 增加过 CPU。当时考虑到它们都是吃 CPU 的大户，只能用一个 CPU 实在太委屈它们了，所以增加了 CPU 数量，想让它们的性能好一些。

可是没想到，ES data 和 ES client 对应用的影响有这么大。

我当时改 ES 的 CPU，是因为我们架构中的一个搜索服务用到了它，而当时的 CPU 给的是一个 C，这导致 Search 服务的 TPS 很低。关于这一点，我在第 15 讲中有描述，你如果不清楚，可以再看看。

同时，我们还要注意一点，ES data 和 ES client 都不是单一的节点，而是有几个节点。由此产生的问题就是，任意一个 ES 节点出现资源消耗过多的时候，都会影响到它所在的 worker 机器资源，进而影响到这个 ES 节点所在的整个物理机。

既然 ES 的进程消耗资源占居首位，那我们该怎么办呢？为了验证问题，我先把 ES 给停掉，看看 TPS 能不能上来，如果能上来，我们再考虑怎么限制 ES 的资源。

停了 ES 之后，TPS 如下图所示：

看到没有？TPS 增加了一倍，并且也没有掉下来！非常理想！

所以，接下来，我们就要考虑把 ES 限制到固定的 worker 上，让它不影响现在的应用。

总结

在这节课中，我们有两个阶段的分析。

在第一个阶段中，我们定位了数据问题。对于性能来说，**数据是非常重要的基础资源，而数据的合理性直接影响了测试的结果。**

经常有初入性能行业的人讨论：性能脚本中的数据到底应该用多少？我觉得这是一个非常明确的问题，**在所有的性能场景中，使用的资源都应该按真实发生的业务逻辑来确定，有且只有这样，才能保证结果是有效的。**

在第二阶段中，我们定位了一个有意思的问题，而这个问题的复杂性在于整体的架构。因为我们是用 KVM、Kubernetes 和 Docker 作为基础设施的，而我们选择的应用原本也不是真正的微服务，是对一个开源系统架构做了更改，把它改成了真正的微服务。

在这样的环境中，如果一个应用有问题，那在修改重启的时候，应用会被 Kubernetes 调度到哪个节点上，是不确定的。也正是出于这样的原因，我们一会儿看到这里有问题，一会儿看到那里有问题，定位的逻辑全都对，但是就是层面不对。

所以，根据我们在第 4 讲中提到的性能分析决策树，我们仍然需要有全局监控、定向监控的思路，并且还要找到计数器的相关性。这样一来，当看到相关计数器有问题的时候，我们就能知道它们之间的关联关系了。

希望你在这节课中，能把性能分析的逻辑记在心中。

课后作业

最后，请你思考一下：

1. 性能脚本中的参数应该如何设计？
2. 如何定位 TPS 会掉下来的情况？大概描述一下你的思路。

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下这节课再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 查询购物车：为什么铺底参数一定要符合真实业务特性？

下一篇 19 | 生成订单信息之一：应用JDBC池优化和内存溢出分析

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。